INSA Rennes – Département informatique Mai 2009



TBXCAST

Un protocole de routage multicast explicite

Rapport final



Cyril BOULEAU
Hamze FARROUKH
Loïc LE HENAFF
Mickaël LECUYER
Jozef LEGENY
Benoît LUCET
Emmanuel THIERRY

Encadrants : Miklós MOLNÁR, Bernard COUSIN

Sommaire

1 Introduction

Ce rapport décrit l'état final du projet TBXcast sous plusieurs angles. Tout d'abord, il contient l'ensemble des informations relatives à la construction du protocole, phase qui a suivi la conception logicielle : découpage en versions, contenu et tests de ces différentes versions. Nous décrirons l'état de la plateforme en cette fin de projet, car elle a été pour nous un outil de test et d'expérimentation indispensable, et constituait ainsi un « petit projet » en elle-même.

Ce rapport fait également office de bilan, aussi bien sur le plan de l'état de finalisation du protocole que sur le plan de l'organisation et de la vie de projet. Nous conclurons quant à l'atteinte des objectifs que nous nous étions posés lors de la planification, et exprimerons l'apport et l'expérience tirés de ce projet.

Le manuel présent en fin de document décrit de manière exhaustive la plateforme d'expérimentation et son mode d'emploi. Un manuel d'utilisation du protocole, c'est-à-dire de sa librairie de haut niveau, figure également. Cette annexe est le fruit d'une année d'expérimentation, de recherche et de développement, et sera précieuse si le projet est amené à être reconduit.

2 Développement de TBXcast

Le protocole TBXcast est un protocole de routage multicast explicite arborescent. Etant donnée sa difficulté de réalisation, nous nous sommes basés sur le code d'un protocole existant, le protocole Xcast. Malgré tout, les spécifications et la conception ont été indépendantes de ce protocole et les fonctionnalités et structures de TBXcast lui sont propres.

2.1 Rappels

Rappelons brièvement la structure du protocole. Les destinataires d'un paquet d'information sont explicitement codés dans l'entête de ce paquet, c'est pour cela qu'il est qualifié « d'explicite ». De plus, la représentation des chemins depuis la source vers les destinataires est un arbre de diffusion multicast, d'où l'appellation « arborescente ».

TBXcast se décompose en plusieurs modules :

- le code noyau, ou « driver TBXcast », qui est la partie présente sur les routeurs intermédiaires. Sa fonction est de faire progresser le paquet sur le réseau en réacheminant les données et en modifiant correctement l'arbre présent dans le paquet
- la librairie LibTBXcast, qui est implémentée à la source et qui a pour rôle la construction de l'arbre en fonction de la topologie et le premier envoi des paquets vers les membres du groupe
- l'application de test TBXTest, qui envoie simplement un paquet à un groupe. Cette application est la seule utilisant TBXcast et sa librairie, et c'est en sniffant le paquet envoyé que nous pouvons réaliser nos tests

Ces trois modules sont donc nécessaires au bon fonctionnement de TBXcast, et ont pu être développés en parallèle. Pour une présentation exhaustive du protocole Xcast et des fonctionnalités du protocole TBXcast, il est possible de se référer aux précédents rapports de spécification et de conception.

TBXcast est développé selon des versions incrémentales, allant de la version 0 à la version 7. Nous avions prévu de réaliser la version 3 cette année. Le précédent rapport de conception détaille de manière complète le contenu de ces versions.

La suite va présenter notre phase de construction, cependant nous avons choisi de clairement scinder la présentation selon les versions. En effet, plutôt que de présenter les objectifs, modifications par rapport à la conception et les difficultés rencontrées pour l'ensemble du projet, nous allons le faire pour chaque version séparément. Cela nous semble judicieux car chaque version représente, en termes de code, des modifications progressives (ajouts et suppressions) apportées à Xcast.

Pour chaque version il y aura donc un rappel de ses objectifs, son état d'avancement, les difficultés rencontrées pendant son développement et enfin son protocole de test accompagné des résultats. Il est également judicieux de noter que chaque version a pour objectif la finalité du protocole. Les fonctionnalités se synthétisent donc entre elles pour donner, à la version 3, un protocole de base qui est censé fonctionner.

2.2 Version 0 : renommage

Cette version est un préliminaire à notre projet et nous sert de base pour le développement de TBXcast. Elle a pour but de renommer l'ensemble du code de Xcast et de sa librairie, LibXcast, pour avoir un protocole identique. Les noms des fonctions, des variables et des constantes doivent être modifiés et préfixés par «tbxcast» au lieu de «xcast». De même, les identifiants de Xcast (identifiant des paquets, adresse multicast globale…) doivent être modifiés pour éviter tout conflit. Les conventions de nommage de Xcast doivent par ailleurs être conservées.

Au final, nous devons obtenir une duplication du protocole Xcast, qui puisse fonctionner en parallèle.

2.2.1 Etat d'avancement

La totalité du driver Xcast a été renommé ainsi que la librairie LibXcast. Certains fichiers systèmes ont également dû être renommés pour intégrer cette version dupliquée de Xcast. La compilation et les tests se sont déroulés avec succès.

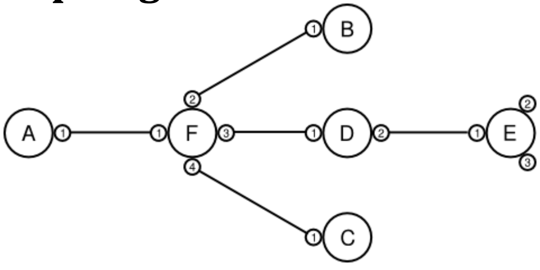
2.2.2 Tests

Les tests de cette version se sont déroulés avec l'application TBXTest. Cette application construit un paquet TBXcast et l'envoie vers un ensemble de destinataires fourni en paramètre. Nous avons construit plusieurs topologies (voir annexe) dont chacune permet de tester une situation type du réseau. A chacune de ces topologies, on associe un ensemble de tests à effectuer.

Résultats

Nous avons choisi de ne tester que sur les topologies 3 et 4.

Topologie 3

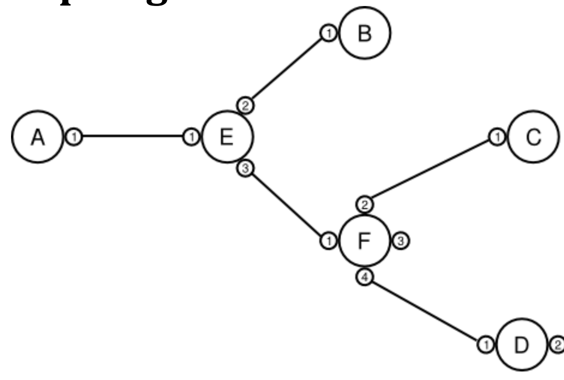


Test : envoi d'un paquet de A vers B et C.
Résultat : réception en F, B et C.

Test : envoi d'un paquet de A vers B et D.
Résultat : réception en F, B et D.

Test : envoi d'un paquet de A vers B, E et C.
Résultat : réception en F, B, D, E et C.

Topologie 4



Test : envoi d'un paquet de A vers B, C et D.
Résultat : réception en E, B, F, C et D.

Test : envoi d'un paquet de A vers E, B, F, C et D.
Résultat : réception en E, B, F, C et D.

Test : envoi d'un paquet de A vers C.
Résultat : réception en E, F et C.

Les résultats obtenus sont conformes à nos attentes.

2.3 Version 1 : implémentation du tunneling

La méthode d'envoi des paquets TBXcast doit généraliser l'utilisation du tunneling classique afin de permettre l'utilisation de TBXcast dans un environnement hétérogène (tous les routeurs ne sont pas forcément compatibles avec TBXcast). Cette version est développée avec anticipation sur la finalité du protocole. Nous supposons donc que l'arbre de routage est bien présent dans l'entête des paquets TBXcast. Nous nous intéressons ici uniquement à la partie concernant l'envoi du paquet, en vue de son acheminement correct sur le réseau.

2.3.1 Rappels sur le tunneling

L'idée[1] était d'utiliser la possibilité d'encapsuler un paquet IPv6 dans un autre paquet IPv6. L'entête du paquet encapsulé est appelé «inner IP header» et le nouvel entête est appelé «outer IP header». Cette technique doit ainsi palier au problème d'hétérogénéité d'un réseau, en considérant que tous les routeurs ne sont pas forcément compatibles avec TBXcast. Ainsi, les routeurs ayant à traiter le paquet en tant que paquet TBXcast sont les routeurs codés dans l'arbre de routage, contenu en entête de chaque paquet TBXcast. Si le paquet passe par d'autres routeurs intermédiaires, ce qui est fort probable dans un environnement de production, alors il sera traité en tant que paquet IPv6 grâce à l'outer IP header.

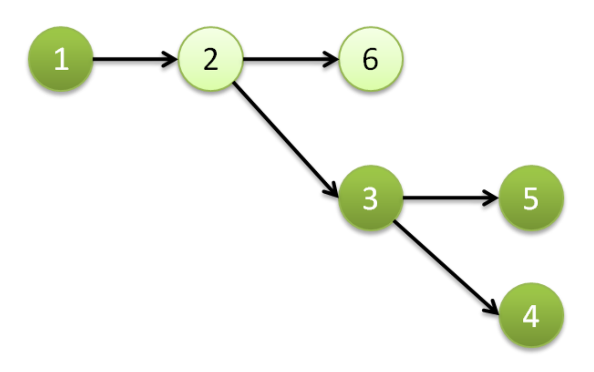


Figure 1 : illustration de l'exemple 1

Outer IP header	Inner IP header	TBXcast header	Transport header
Next header : IPv6 **Source Adress : 1** **Destination Address : 3**	Next header : Routing header (TBXcast) Source Adress: 1 Destination Address : TBXcast_all_routers	Codage de l'arbre de routage.	TCP / UDP

Tableau 1 : structure d'un paquet circulant sur le réseau décrit par la Figure 1

Sur la figure 1 : Les cercles foncés représentent les routeurs TBXcast codés dans l'arbre. Les cercles clairs sont de simples routeurs intermédiaires sur le réseau, non compatibles TBXcast. L'outer IP header représente l'entête permettant de mettre en œuvre le tunnel. Il a pour effet d'englober le paquet à partir de l'inner IP header afin qu'il se comporte comme un paquet IPv6 entre deux nœuds de l'arbre de routage. Sur l'exemple, le paquet est traité comme un paquet IPv6 entre les routeurs 1 et 3. C'est à dire que le routeur 2 n'aura pas conscience qu'il s'agit d'un paquet TBXcast en réalité et le transmettra normalement au routeur 3. Comme nous voulons généraliser l'utilisation du tunneling, une fois que le paquet sera arrivé au routeur 3, celui-ci sera traité par le driver TBXcast, sera empaqueté une nouvelle fois par un nouvel outer IP header pour être transmis au prochain routeur codé dans l'arbre de routage (ici le routeur 4 par exemple). Le nouvel entête sera alors comme illustré ci-dessous. On remarquera également que s'il n'y a pas de routeurs non compatibles TBXcast entre deux nœuds, le paquet est quand même encapsulé et un outer IP header est ajouté.

[1] Pour plus de détails techniques, voir le rapport de conception, chapitre 4 : routage d'un paquet TBXcast

Outer IP header	Inner IP header	TBXcast header	Transport header
Next header : IPv6 **Source Adress : 3** **Destination Address : 4**	Next header : Routing header (TBXcast) Source Adress: 1 Destination Address : TBXcast_all_routers	Codage de l'arbre de routage.	TCP / UDP

Tableau 2 : la nouvelle structure du paquet

2.3.2 Difficultés rencontrées et solutions

En théorie assez simple, l'implémentation du tunneling tel que décrit précédemment s'est avérée très difficile et s'est soldée par un changement d'objectif. En effet, Xcast utilise le «semi-permeable tunneling», technique proche de la nôtre mais implémentée différemment au final. Ce type de tunneling met en jeu un entête supplémentaire entre l'inner IP header et l'outer IP header, nommé entête « Hop-by-Hop ». Cet entête est lu par tous les routeurs, et a pour objectif de déterminer si un routeur a la capacité de lire et d'interpréter le contenu du paquet. En l'occurrence, un champ de cet entête indique que la suite du paquet est de type Xcast et que seuls les routeurs implémentant le driver Xcast peuvent traiter ce paquet comme il se doit. Les autres routeurs le traiteront en tant que paquet IPv6 classique grâce à l'outer IP header. La destination du paquet encapsulée est la première adresse des destinations du paquet Xcast. Ainsi, en l'absence de routeur compatible avec Xcast, le paquet poursuit son chemin vers la première destination. Si au cours du routage, le paquet est reçu par un routeur compatible avec Xcast, alors le paquet est décapsulé et traité par le driver Xcast.

L'implémentation du semi-permeable tunneling de Xcast s'est avérée en réalité très minutieuse et étroitement liée aux fichiers systèmes de NetBSD. Le manque de temps et la difficulté de la chose nous ont poussé, à ce jour, à conserver le tunneling tel qu'il est implémenté dans Xcast.

Ce dernier choix a amené un nouveau problème. En effet, sur une route, l'entête Hop-by-Hop est lu par tous les routeurs pour savoir si la suite du paquet est interprétable. Dans notre cas, sur une route donnée, tous les routeurs compatibles avec TBXcast vont alors traiter le paquet. On peut alors imaginer une topologie de réseau où il y a de nombreux routeurs TBXcast et où une petite partie des routeurs seulement est codée dans l'arbre de routage d'un paquet. Rappelons que seuls les routeurs TBXcast codés dans l'arbre de routage du paquet doivent traiter le paquet en tant que paquet TBXcast. Avec l'entête Hop-by-Hop, tous les routeurs compatibles TBXcast qui se trouvent sur le chemin entre deux nœuds de l'arbre vont également traiter le paquet avec le driver TBXcast. Cela peut à terme se solder par un changement de route, différente de celle codée dans l'arbre de routage du paquet.

Ce dernier problème nous a donc amené à changer nos hypothèses de travail. A savoir, nous considérerons pour le moment que tous les routeurs TBXcast présents sur une route seront codés dans l'arbre de routage.

2.3.3 Etat d'avancement

Le semi-permeable tunneling de Xcast a donc été conservé. Seul le champ destinataire de l'outer IP header a été modifié pour qu'il corresponde au prochain routeur TBXcast de l'arbre de routage, et non plus à un destinataire au hasard, comme c'était le cas dans Xcast. Il s'agit ici d'une simple adaptation à notre structure d'arbre.

La version 1 est donc fonctionnelle mais est soumise à une hypothèse forte que nous n'avions pas envisagée lors de la conception.

2.3.4 Tests

Etant donné que nous conservons à de petits détails près le tunneling codé dans Xcast, les tests se sont basés sur la version 0 et se sont soldés par un succès.

2.4 Version 2

Le but de la version 2 est d'implémenter la structure de l'arbre finale dans le code. On fournit la structure de l'arbre complète à la source. Dans les routeurs, le driver est modifié pour prendre en compte un nouvel algorithme de routage.

2.4.1 Mise à jour depuis la conception

Dans le driver, il existe une principale modification que nous avons effectués depuis le rapport de conception.

Contrôle

Dans la version 2, nous avions fait le choix de supprimer les vérifications effectués sur le paquet. Ceci permet de vérifier le bon fonctionnement de l'algorithme de routage sans qu'il soit perturbé par d'éventuelles erreurs de contrôle. Pour ce faire, nos tests ne gèrent pas les cas de mauvaise utilisation pour le moment (problèmes de routeur, mauvaises adresses, etc.). Ces contrôles seront à remettre dans les versions ultérieures.

2.4.2 Etat d'avancement

2.4.2.1 Librairie

La librairie, qui en est sa version 2, permet l'envoi des messages en utilisant un arbre fourni par l'utilisateur. Cet arbre peut être soit construit directement dans le code ou alors chargé à partir d'un fichier binaire.

Outils

Afin de faciliter la création de ces fichiers binaires nous avons développé un script qui les produit à partir des fichiers XML.

Par exemple ce fichier XML :

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<tree>
	<router address="FF::1">
		<router address="FF::2">
			<dest address="FF::4" dest="1"/>
			<router address="FF::5">
				<dest address="FF::7"/>
			</router>
		</router>

		<router address="FF::3" dest="1">
			<router address="FF::6">
				<dest address="FF::8"/>
			</router>
		</router>
	</router>
</tree>
```

Produira l'arbre de routage suivant :

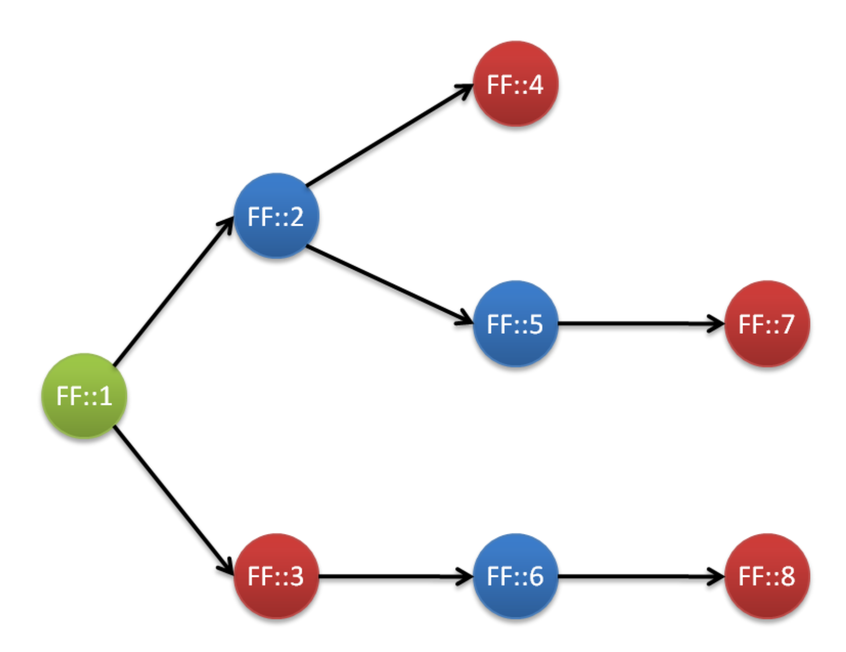


Figure 2 : arbre correspond à la description XML

Modifications

Structures

Cette version de TBXcast n'utilise plus une liste de destinataires à plat comme Xcast mais l'arbre de routage comme défini dans les spécifications de TBXcast[2]. Ce changement se reflète dans la librairie par le changement de la structure contenant les informations sur un groupe TBXcast.

Fonctions

Cette version de la librairie ajoute donc 2 nouvelles fonctions.

- *TBXcastReadTree* – Lit un fichier contenant la description de l'arbre et crée la structure C correspondante.
- *TBXcastAddTree* – Associe un arbre de routage à un groupe.

Régressions par rapport à la version précédente

Vu la nature du routage arborescent, il n'est plus possible de simplement ajouter un membre à un groupe car l'arbre doit être recalculé à chaque fois. Toutes les fonctions de gestion des membres sont alors supprimées en attendant la version 3.

[2] Se référer à la partie 3.3.2 du rapport de conception

2.4.2.2 Driver

Le protocole intègre une nouvelle structure appelée tbxcast6_node contenant un champ longueur et destinataire et un champ pour l'adresse IPv6. Pour compléter cette structure, l'ajout de deux macros a été nécessaire. La première permet de récupérer le champ longueur de la structure quant à l'autre, elle permet de récupérer le champ destinataire.

```
typedef struct tbxcast6_node {
      uint8_t          tbx6n_lgbm; /*  length  of  the  sub-tree  and
bitmap on the lowest bit */
      struct in6_addr   tbx6n_addr; /* IPv6 address */
} tbxcast6_node_t;

#define TBXCAST6_NODE_SUBTREELEN(tbx6n_lgbm)    \
      (tbx6n_lgbm >> 1)

#define TBXCAST6_NODE_ISDEST(tbx6n_lgbm)  \
      (tbx6n_lgbm & 1)
```

Certaines macros servant au calcul de longueur de l'ancien bitmap de Xcast ne servent plus. Par contre, de nouvelles macros pour des calculs similaires sont désormais disponibles mais non présentées ici. Elles sont commentées et définis dans le même fichier que la structure de nœud.

2.4.3 Tests

Etant donné que les changements pour passer de la version 2 à la version 3 ne concernent que la librairie, et que la version 2 de celle-ci n'est pas la mieux adaptée pour faire des tests, on a préféré directement travailler à partir de la librairie de la version 3. Néanmoins toutes les fonctions de manipulation de l'arbre ont été testées de manière unitaire. Voir la partie « test » de la version 3 ci-dessous (2.5.5).

2.5 Version 3 : création de l'arbre à partir de la topologie

Cette version est la dernière version prévue pour cette année. Elle a comme objectif de créer l'arbre de routage à la source à partir de la topologie du réseau pour ensuite construire le paquet TBXcast. Ce paquet sera ensuite transmis grâce aux fonctions développées à la version 2.

A la fin de cette version, nous avons un protocole opérationnel qui peut transmettre des paquets depuis la source jusqu'aux destinataires. Il ne reste que la topologie du réseau qui n'est pas gérée automatiquement. La topologie devra être apportée manuellement à la source qui pourra alors construire l'arbre de routage vers les destinataires souhaités.

2.5.1 Mise à jour depuis la conception

L'algorithme mis en place lors de la conception pour créer l'arbre de routage a nécessité quelques traitements supplémentaires en amont et en aval pour pouvoir être implémenté avec les structures dont nous disposons.

Pour rappel, lors de la conception nous avons donné un algorithme qui permet de créer l'arbre de routage en connaissant la topologie complète et les destinataires. Cet algorithme s'appuie sur l'algorithme de Moore-Dijkstra et nous renvoie un arbre des plus courts chemins entre la source et les destinataires.

Dans cet algorithme, la topologie est représentée sous la forme d'une matrice contenant à la case (i, j) 1 s'il y a un lien entre les nœuds i et j et 0 s'il n'y en a pas. Le problème est que concrètement, la topologie n'est pas représentée par des liens entre des nœuds mais par des liens entre les interfaces réseaux présentent sur ces nœuds.

Nous avons donc rajouté une structure permettant de représenter cela ; cette structure contient trois champs :

- le numéro du nœud de l'extrémité source du lien,
- le numéro du nœud de l'extrémité destinataire du lien,
- l'adresse de l'interface destinataire.

Par exemple, le lien de la Figure 3 ci-dessous sera représenté de la manière suivante :

{ source = 0, destinataire = 1, adresse = b::1 }

{ source = 1, destinataire = 0, adresse = a::1 }

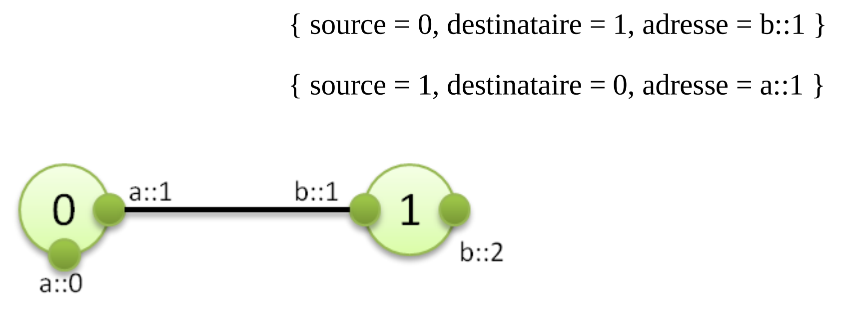


Figure 3

Ces structures sont ensuite regroupées dans un tableau représentant alors la topologie complète du réseau.

Cette représentation de la topologie est beaucoup plus proche de la représentation réelle de la topologie que l'on pourrait récupérer avec des protocoles comme OSPF par exemple, ce qui est plus adapté aux futures versions du protocole. Il est, de plus, facile de retrouver la matrice que l'on avait précédemment en faisant un simple parcours du tableau et en rajoutant un 1 à la case

(source, destination). On peut de même récupérer les numéros de nœud des destinataires à partir de leurs adresses simplement en cherchant l'adresse dans la topologie, le numéro du nœud est alors celui du destinataire.

Ce prétraitement effectué, on peut alors appliquer l'algorithme de Moore-Dijkstra. Cet algorithme renvoie un arbre sous la forme d'un tableau de prédécesseurs et qui peut contenir des branches inutiles. On doit alors élaguer l'arbre. Pour cela, il suffit de partir des destinataires et de remonter vers la racine en suivant les prédécesseurs. Les branches non parcourues ainsi ne contiennent pas de destinataires et sont donc supprimées de l'arbre. On obtient alors le réel arbre des plus courts chemins.

On effectue ensuite un traitement supplémentaire qui consiste à supprimer les nœuds inutiles pour le routage. En effet, si nous avons une branche telle que représentée dans la Figure 4, on peut alors supprimer le nœud 1 car le nœud 0 peut envoyer le paquet directement au nœud 2 en passant par un tunnel unicast.



Figure 4

Pour supprimer ces nœuds, on compte le nombre de successeurs à chaque nœud, on supprime alors les nœuds qui n'ont qu'un seul successeur et s'ils ne sont pas destinataires.

On doit enfin effectuer un dernier traitement qui consiste à transformer l'arbre rendu par l'algorithme en un arbre TBXcast. On peut pour cela récupérer les adresses dans la topologie que nous avons effectuée.

2.5.2 Etat d'avancement

A l'heure actuelle, la version 3 est fonctionnelle depuis la création de la topologie jusqu'à la création de l'arbre. Nous avons pu tester les différentes étapes de l'algorithme grâce au protocole de test décrit dans la partie correspondante.

Nous n'avons cependant pas pu tester si l'arbre est bien inclus dans le paquet TBXcast que l'on veut envoyer. En effet, la version 2 n'étant pas encore fonctionnelle, nous ne pouvons pas tester l'envoi du paquet après création de l'arbre. Nous ne pouvons que supposer que la structure de l'arbre est bien intégrée dans le paquet.

Les décisions prises à la version 1 nous ont également obligés à modifier légèrement l'algorithme : nous avons dû désactiver l'étape de suppression des nœuds inutiles au routage. En effet nous avons dû supposer à la version 1 que tous les routeurs TBXcast situés entre la source et les destinataires sont présents dans l'arbre, nous ne pouvons donc plus supprimer de nœuds dans l'arbre.

2.5.3 Difficultés rencontrées et solutions

Lors de la réalisation de cette version, nous avons rencontré deux principales difficultés, la première liée à la transformation de l'arbre de l'algorithme en arbre TBXcast et la deuxième liée à l'intégration de l'algorithme dans le code existant de la librairie de Xcast.

Le premier problème est dû au fait que l'algorithme de Moore-Dijkstra que nous avons utilisé est adapté à une représentation de l'arbre complètement différente de la notre. En effet, l'arbre est représenté dans l'algorithme à l'aide d'un tableau des prédécesseurs. La case i de ce tableau contient le prédécesseur du nœud i dans l'arbre ou -1 s'il n'a pas de prédécesseur (s'il n'est pas dans l'arbre ou si c'est la racine).

Cette représentation diffère totalement de la représentation que nous avons utilisée et adapter l'algorithme dont nous disposions à notre représentation se serait avéré trop compliqué. Nous avons

donc effectué un algorithme qui transforme l'arbre sous la première représentation en arbre TBXcast. Cet algorithme est présenté ci-dessous :

```
src_i : indice de la source dans l'arbre de l'algorithme
pred(i) : prédécesseur du nœud i récupéré par l'algorithme
nœuds(j) : indice du nœud j de l'arbre TBXcast dans l'arbre de l'algorithme
pred_courant : pile des prédécesseurs du nœud courant de l'arbre TBXcast
p : indice du sommet de pile
lg(j) : longueur associée au nœud j de l'arbre TBXcast
/* on ne met pas le bit de destinataire et l'adresse associée au nœud j car
ils ne sont pas utiles à l'algorithme.
Pour mettre le bit de destinataire, il suffit de regarder si le nœud qu'on
ajoute est présent dans la liste des destinataires donnée en entrée, les
adresses des nœuds sont présentes dans la topologie */

// initialisation
nœuds(0) := src_i
lg(0) := 1
p := 0
pred_courant[p] := 0
i := 0

// algorithme
tant que i < nombre de nœuds dans l'arbre TBXcast faire
      chercher le successeur n de nœuds(pred_courant[p]) non encore mis
dans nœuds
      si on a trouvé un nœud n alors
            nœuds(i) := n
            lg(i) := 1
            pour tout k de 0 à p faire
                  lg(pred_courant[p]) := lg(pred_courant[p]) + 1
            fait
            pred_courant[p] := i
            i := i+1
      sinon
            p := p-1
      fsi
fait
```

La deuxième difficulté de cette version a été d'intégrer l'algorithme réalisé dans le code de Xcast. En effet la librairie héritée de Xcast représente un groupe de destinataires par une structure assez compliquée contenant les informations nécessaires à la construction du paquet Xcast. Ces informations contiennent la liste des adresses et le bitmap propres au protocole Xcast.

Pour intégrer l'arbre de routage dans cette structure nous avons remplacé ces deux éléments propres à Xcast par notre arbre.

Le problème est que ces deux éléments (liste des destinataires et bitmap) son régulièrement utilisés dans les fonctions de la librairie. Notamment, lors de l'ajout d'un membre, ces structures sont réallouées pour pouvoir ajouter l'adresse de celui-ci dans la liste. Ces réallocations entrainent un certain nombre de calculs sur les informations contenues dans la structure de groupe (taille de l'entête du paquet, etc.) et s'avèrent assez compliqués à remplacer pour tenir compte de l'arbre de routage.

Nous nous sommes finalement rendu compte que la politique adoptée par Xcast lors de l'ajout d'un membre n'est plus réellement applicable à notre version du protocole. Nous avons donc décidé de complètement changer la représentation d'un membre dans la librairie.

En réalité nous avons mis en place un tableau qui contient toutes les adresses des membres des différents groupes et qui leur associe les identificateurs des groupes dans lesquels ils sont présents. Ainsi lorsqu'on ajoute un membre dans un groupe, il suffit de rajouter une entrée dans la table. De

plus, on a mis en place une fonction qui permet de retrouver tous les membres d'un groupe en lisant ce tableau. Ainsi, la fonction créant l'arbre peut avoir accès à la liste des membres juste en ayant l'identificateur du groupe pour lequel on veut créer l'arbre.

Nous avons réalisé pour la version 2 une fonction permettant d'associer un arbre à un groupe et qui fait tous les calculs de changement de taille des données, nous pouvons réutiliser cette fonction une fois l'arbre créé.

Cela change totalement la politique d'utilisation de la librairie vis-à-vis de l'utilisateur. Avec Xcast, un utilisateur pouvait créer un groupe et lui ajouter des membres, les structures du groupes étaient alors mises à jours lors de l'ajout de chaque membre. Avec les modifications que nous avons apportées pour TBXcast, un utilisateur pourra créer un groupe et lui ajouter des membres. Par contre ce sera à lui de calculer l'arbre et d'ajouter l'arbre dans le groupe en faisant explicitement appel aux fonctions correspondantes. La gestion des membres et celle de l'arbre sont donc complètement indépendantes. L'utilisateur peut calculer l'arbre quand il le souhaite, ce qui est plutôt un avantage car l'algorithme de calcul de l'arbre est assez lourd.

2.5.4 Tests

Etant donné que les tests avec la version 3 de la librairie sont plus faciles (la création des fichiers représentant les arbres n'est plus nécessaire), nous avons donc décidés d'utiliser celle-ci pour tester la version 2 du driver.

Nous avons décidé de séparer les tests de la librairie de ceux du driver. Nous avons donc réalisé un autre programme de test qui ne fait que les tests utiles à cette version. L'avantage est que nous avons pu coder la topologie que l'on souhaitait sans être obligés de la mettre en place dans la salle d'expérimentation. Nous pouvons aussi changer la source du paquet comme on le souhaite sans changer de poste de test.

Nous ne pouvons pas encore effectuer de tests sur le driver car il reste encore des problèmes lors de l'envoi du paquet de la librairie vers le driver.

2.5.4.1 Protocole de test

Le fichier que nous avons créé pour tester la version 3 a pour but de tester toutes les modifications que nous avons apportées à la librairie. Nous avons donc fait appel à toutes les fonctions modifiées et ajoutées dont nous avons vérifié le bon fonctionnement.

Le test que nous avons mis en place va donc créer un groupe et lui ajouter des membres. Nous vérifions ensuite que les membres ont bien été ajoutés.

On créé ensuite la topologie en ajoutant successivement les liens entre les nœuds. Cette topologie est ensuite vérifiée en affichant la totalité de celle-ci ainsi que la matrice de routage de l'algorithme.

On fait ensuite appel à la fonction de création de l'arbre lors de laquelle on vérifie les indices des nœuds de la source et des membres dans la topologie.

On vérifie enfin que l'arbre créé est bien celui demandé.

2.5.4.2 Résultat des tests

Nous avons mis en place pour les différents tests effectués, la topologie suivante :

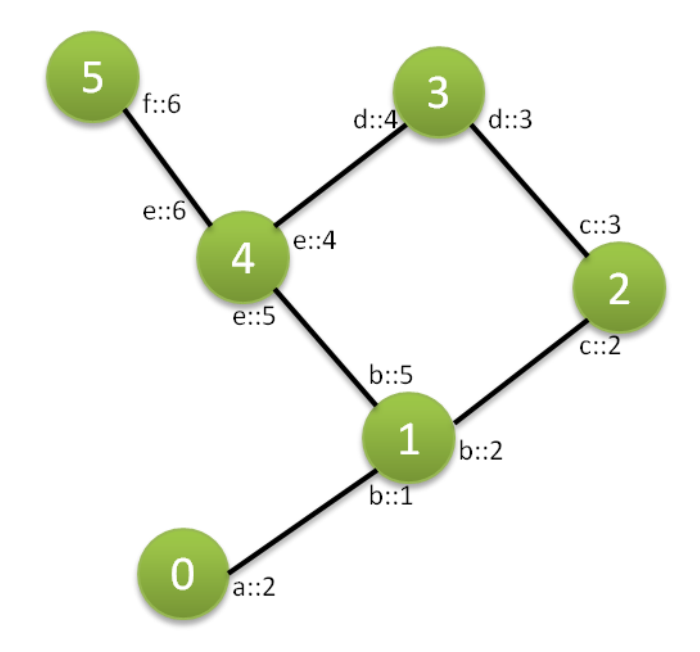


Figure 5 : topologie de test

Nous avons explicitement mis une boucle dans la topologie afin de vérifier le comportement de l'algorithme.

La topologie affichée par le protocole de test est la suivante :

```
Table de routage
S : 5, D : 4, addr : e::6
S : 3, D : 4, addr : e::4
S : 3, D : 2, addr : c::3
S : 4, D : 5, addr : f::6
S : 1, D : 4, addr : e::5
S : 4, D : 3, addr : d::4
S : 2, D : 3, addr : d::3
S : 4, D : 1, addr : b::5
S : 2, D : 1, addr : b::2
S : 0, D : 1, addr : b::1
S : 1, D : 2, addr : c::2
S : 1, D : 0, addr : a::1
```

Cette topologie correspond bien à celle demandée.

La matrice créée est la suivante :

```
    0  1  2  3  4  5
0 : 0  1  0  0  0  0
1 : 1  0  1  0  1  0
2 : 0  1  0  1  0  0
3 : 0  0  1  0  1  0
4 : 0  1  0  1  0  1
5 : 0  0  0  0  1  0
```

Elle correspond également à la topologie.

Dans la suite nous allons afficher les arbres résultant des différents appels que nous avons effectués. Nous afficherons les arbres tels que la version actuelle les crée, c'est-à-dire sans supprimer les nœuds inutiles au routage. On peut facilement imaginer les arbres réalisés avec la version complète de l'algorithme.

Nous avons ensuite réalisé plusieurs tests d'arbre :

- envoi de a::1 à f::6 et c::2 (test d'un arbre simple) :

  Le résultat affiché est le suivant :

```
source : 0
affichage des destinataires
5 2
Nbr d'adresses elagues 5
taille de l'arbre : 5
0  -  lg : 5 , d : 0, addr : a::1
1  -  lg : 4 , d : 0, addr : b::1
2  -  lg : 1 , d : 1, addr : c::2
3  -  lg : 2 , d : 0, addr : e::5
4  -  lg : 1 , d : 1, addr : f::6
```

  Ce qui correspond à l'arbre suivant, qui est bien l'arbre voulu.

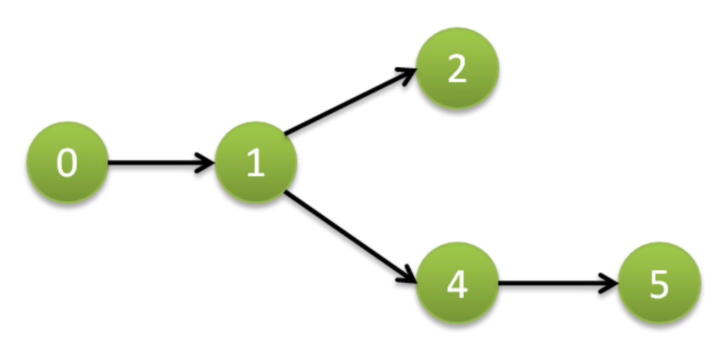


Figure 6 : arbre résultat pour le premier envoi

- Envoi de a::1 à e::4 (test des boucles)

  Le résultat affiché est le suivant :

```
source : 0
affichage des destinataires
4
taille de l'arbre : 3
0  -  lg : 3 , d : 0, addr : a::1
1  -  lg : 2 , d : 0, addr : b::1
2  -  lg : 1 , d : 1, addr : e::5
```

  Ce qui correspond à l'arbre suivant, qui est bien l'arbre voulu.



Figure 7 : arbre résultat pour le second envoi

  On note que l'adresse associée au troisième nœud est e::5 et non e::4. Cela correspond à nos attentes car ces deux interfaces sont situées sur le même nœud et l'algorithme renvoie bien le chemin le plus court qui aboutit à une autre interface que celle demandée mais bien au bon destinataire.

Comme annoncé dans le rapport de conception nous nous sommes arrêtés à la version 3 cette année. Nous allons maintenant présenter notre plateforme d'expérimentation qui nous a permis de tester ces versions du protocole.

2.6 Plateforme d'expérimentation

Comme nous l'indiquions dans les premiers rapports, pour les besoins des tests, une plateforme de test est indispensable. Cela a été la motivation de sa mise en place. Nous avons récupéré et fait évoluer la plateforme de l'année dernière.

2.6.1 Objectifs fixés en début de projet

La plateforme de l'année dernière a été constituée de machines hétérogènes et ne comportait pas de programmes de gestion, ni aucune procédure d'automatisation. Cette absence de gestion n'est pas viable. Comme nous le montre le manuel d'installation de TBXcast (cf Wiki), la compilation du noyau peut prendre une dizaine de minutes, et demande des dizaines de commandes. Toutes ces commandes doivent être faites sur toutes les machines de la plateforme. Le problème est le même pour la configuration du réseau IPv6. Soit on choisit de définir cette topologie de manière fixe dans la configuration du système, et dans ce cas, on ne peut plus en changer. On perd alors énormément de flexibilité, puisque l'on ne peut pas tester des topologies diverses. Soit on choisit d'effectuer cette configuration de manière manuelle. Et dans ce cas, à chaque changement de topologie, et à chaque redémarrage des machines, la procédure demande des commandes supplémentaires sur les machines, et sur le switch. Il faut donc refaire la configuration du réseau au moins aussi souvent que l'on change de noyau.

Ce mode de fonctionnement n'est pas envisageable pour la réalisation d'un protocole de routage. Ce genre d'applications est difficile à mettre en œuvre et demande un nombre conséquent de tests. Et à chaque test, ce sont en définitive des centaines de commandes à taper.

Il nous fallait un cadre, des procédures pour faciliter la gestion, automatiser la préparation des tests. C'était l'objectif de tout le travail sur la plateforme.

Après quelques études, il fut primordial de commencer par homogénéiser les configurations matérielles. Il est très complexe de gérer des automatisations sur des configurations différentes, puisque cela peut demander des cas particuliers. On peut donner l'exemple des cartes réseaux. Des cartes réseaux différentes auront, sous NetBSD, des noms différents, et il faudra prendre en compte cette différence dans un programme qui gérerait le réseau.

La difficulté du travail a été liée à la maintenance de plusieurs copies du système. Sur une plateforme classique, chaque machine a son propre système, installé sur le disque dur. Et la compilation du noyau doit se faire sur chaque poste.

Enfin la dernière étape était la gestion de la topologie, qui devait se faire le plus facilement possible. Il était donc nécessaire de trouver le moyen de configurer le réseau par un script.

Tous ces objectifs étaient initialement des outils en marge du projet, mais leur importance est telle qu'ils ont finalement totalement été intégré dans le projet. C'est en quelque sorte un projet dans le projet.

2.6.2 Etat d'avancement

Pour parvenir à nos objectifs, nous avons donc commencé par demander le renouvellement total du parc de machines. Nous avons tout d'abord obtenu 5 machines, puis, par des demandes successives, nous sommes parvenus à obtenir 3 machines supplémentaires de la même configuration.

L'organisation que nous avons planifiée initialement prévoyait 7 postes dédiés aux tests, utilisant 14 cartes réseaux, et un poste faisant office de serveur. Par manque de matériel, nous n'avons pas utilisé l'un des postes dans le réseau de test, mais nous l'avons reconverti en serveur. L'architecture finale compte donc 6 machines dédiées aux tests, avec 12 cartes réseaux, 1 serveur, et 1 serveur de secours.

Pour les besoins de la compilation du système, une première idée était d'utiliser une clé USB et des scripts de compilation. Le noyau pouvait n'être compilé qu'une fois, et ensuite copié sur chaque

machine. Cette idée apportait une première automatisation, mais n'était pas encore assez satisfaisante puisqu'elle imposait toujours une gestion totalement indépendante de chaque système. Elle a très rapidement été abandonnée au profit du principe de netboot (démarrage par le réseau).

Cette solution n'a pas mis beaucoup de temps à s'imposer. C'est la réponse logique à la problématique. On veut que l'administration du système soit centralisée, il faut donc centraliser le système d'exploitation sur un serveur, et le servir à chaque machine. La mise en place de cette idée a été longue et complexe. Après la mise en place du service, le système classique a été abandonné et tous les postes ont été migrés sur le système netboot.

Concernant l'automatisation de la compilation et de la gestion de la plateforme, nous avons réalisé 2 programmes et une collection de Makefile.

Les Makefile ont pour but logique de contrôler toute la compilation. Ainsi, le make effectue en une commande toutes les opérations nécessaires à la compilation du noyau et/ou de la librairie et du programme de test. Ensuite, en une commande aussi, on installe les produits compilés sur toutes les machines. Un système de versioning sommaire permet notamment de mettre en cache les compilations des noyaux et de ne pas avoir à les recompiler lorsque l'on veut revenir à une version antérieure.

Un programme, *tbxpwr*, s'occupe des opérations sur le démarrage/arrêt des machines. Il permet de choisir des machines à démarrer, arrêter, ou redémarrer, et de vérifier l'état des machines de la plateforme. Le redémarrage et l'arrêt d'une machine sont implémentés par une simple connexion ssh. Le démarrage d'une machine déjà arrêtée a demandé plus de travail. Il a fallu implémenter un Wake On Lan. Lorsqu'une carte réseau pour laquelle le Wake On Lan est activé, reçoit une certaine trame, elle démarre la machine. Cette technique est importante pour le travail à distance, puisqu'elle permet, sans accès physique à la plateforme, d'allumer et d'éteindre les postes à souhait et non de les laisser allumés en permanence. Seul le serveur doit rester allumé, pour transmettre les commandes.

Un second programme, *tbxnet*, gère la connexion entre les machines. On peut lier ou délier deux machines entre elles si c'est possible, on peut aussi charger une topologie entière si elle a été prédéfinie, et enfin on peut visionner l'état des liens entre les postes. Ces actions se réalisent avec des commandes ssh sur les machines et sur le switch, et récupèrent les informations par le parsing du résultat de la commande *ifconfig*.

Au fur et à mesure de l'avancement de la plateforme, il est devenu de plus en plus indispensable de dissocier un réseau IPv4 entièrement dédié à la gestion, et un réseau IPv6 destiné aux tests. Cette dissociation avait l'inconvénient majeur de nous priver d'une interface par poste, puisqu'une interface devait être branchée sur le réseau ipv4. Mais les besoins sont tels que nous préférions sacrifier quelques possibilités de topologies pour rendre possible l'automatisation.

Pour plus de clarté, toutes les machines ont, dès le départ, été connectées au réseau de gestion par leur interface réseau intégrée (intégrée à la carte mère). Ce détail s'est montré très important :

- Les systèmes de netboot et de Wake on Lan ne peuvent fonctionner que sur la carte intégrée.
- Les cartes intégrées sont indissociables de la machine. Même en cas de réorganisation des cartes réseaux dans les machines, celles-ci ne changent pas d'identité, il n'y a pas à refaire les configurations des différents services.
- Le réseau de test n'utilise qu'un seul et même type de carte réseau (les 12 cartes filles qui nous ont été fournies), ce qui facilite l'écriture des scripts.

2.6.3 Difficultés rencontrées et solutions apportées

2.6.3.1 Technologie Netboot

La technologie Netboot est difficile à appréhender. Il a fallu beaucoup de recherches et de tests pour la mettre en place. Les mécanismes demandent la configuration de plusieurs services, et dont le débogage est délicat.

Tout d'abord il faut configurer le BIOS de chaque machine pour demander le système via le réseau. Il faut aussi configurer plusieurs démons sur le serveur : dhcpd, tfpd, inetd, nfsd, mountd, rpcbind. Il a fallu aussi se familiariser avec le fonctionnement des services sous NetBSD. Les erreurs fournies par le BIOS et le programme de démarrage sont peu explicites.

Il y a notamment eu des problèmes avec le serveur TFTP et le serveur NFS. La plupart du débogage s'est fait de manière unitaire, pour vérifier le bon fonctionnement de ces services, avant de pouvoir les utiliser au sein du netboot.

2.6.3.2 Programmes d'automatisation

Il était initialement prévu de regrouper toutes les procédures dans un seul programme. Dans la mesure où la compilation et la définition de la topologie, par exemple, sont totalement différentes, nous avons scindé les opérations dans plusieurs programmes. La programmation s'en trouve grandement simplifiée, puisque l'on remplace un programme complexe par une suite logicielle. Cette séparation a permis d'implémenter la compilation par un système de Makefile, beaucoup plus adaptée à cette problématique qu'un binaire ou un script.

2.6.3.3 Wake On Lan

Le Wake On Lan nécessite l'envoi d'un format très précis de trames Ethernet. Le format en lui-même est parfaitement défini et documenté. Mais la méthode d'envoi d'une trame est plutôt floue. Tout d'abord, ce n'est pas standardisé mais parfaitement dépendant du système, ensuite la documentation est très peu fournie, même dans les manuels Unix « man » ou sur internet. Après de longues recherches et de nombreux tests, nous avons finalement eu l'idée de générer la trame et de l'envoyer directement par un device. La lecture de la librairie pcap a confirmé cette idée.

A partir de là, le développement du Wake On Lan s'est fait très simplement. Nous avons donc pu développer une première version sur Mac OS, puis la porter sous NetBSD.

2.6.3.4 Forking et structure du programme

Ce qui a pris le plus de temps a été de définir la structure du programme. La très grosse majorité des opérations se basait sur le même principe : l'envoi à plusieurs machines de commandes par SSH. Il fallait donc définir un cadre commun pour éviter de coder plusieurs fois les mêmes opérations. La partie structure a mis longtemps à se fixer, quelquefois trop stricte, d'autres fois trop peu puissante.

Il y a eu plusieurs versions de cette structure. Au départ, on considérait que le programme sur le serveur ne servait qu'à appeler un script sur chaque poste, lequel script s'occupait de faire toutes les opérations. Mais ce modèle était manifestement limité. Chaque opération devait se baser sur un accès, alors qu'en réalité certaines opérations sont purement locales. Il empêchait toute interaction entre les machines, puisque sur chaque machine le script appelé devait avoir directement toutes les informations nécessaires.

Après quelques évolutions, il a été décidé de dissocier la connexion SSH de la structure, et ensuite qu'une commande n'invoquait pas systématiquement un fork du programme. Après redéfinition, la structure est devenue beaucoup plus flexible, constituée de plusieurs librairies :

- une librairie pour le traitement de la commande, qui scanne les arguments de la ligne de commande pour en extraire la sous-commande à exécuter, qui peut afficher l'aide, ou bien les instructions d'usage du programme
- une librairie de fonctions pour l'appel à ssh, qui ne sert qu'à simplifier les appels à la librairie libssh2
- une librairie de données de la plateforme, pour la traduction des noms d'hôte, ou bien le stockage des adresses mac.
- une librairie de forking, qui exécute une fonction dans un fork et en retourne le résultat

Malgré quelques bugs, cette structure est restée plutôt stable et adaptée à tous les usages. Elle a été utilisée pour les deux programmes tbxpwr et tbxnet.

2.6.3.5 Difficultés générales

Plus généralement, les difficultés de mise en place de la plateforme étaient dues à la difficulté de prise en main d'un système BSD, à des manques de documentation, ou à un challenge technologique permanent.

2.6.3.6 Absence de spécification

Une difficulté du développement de la plateforme, qui est en même temps un point fort, se situe dans l'absence de spécification. Il n'y a pas eu d'étude, le développement ne s'est pas fait par un cycle en V, mais par un cycle en spirale. Dans un sens, c'est un fonctionnement adapté. Ce développement gardait un objectif essentiel, évoluer constamment vers une simplification et une automatisation de la plateforme. Chaque nouvelle évolution donnait de nouvelles perspectives de développement, pour tendre vers le but.

Cependant, l'absence de spécifications, pour la plateforme, a rendu certains développements plus complexes.

2.6.4 Bilan et suggestions

Le développement de la plateforme a plutôt bien atteint son objectif. La compilation peut se faire en une commande, on peut gérer complètement le démarrage et l'arrêt des machines. On peut définir des topologies pour les tests. En quelques commandes, on peut préparer la plateforme complète pour un test.

Cependant les programmes de gestion sont encore imparfaits. Certaines fonctionnalités ont été implémentées avec des technologies inadaptées. On aurait pu préférer des threads à des fork, ou bien des appels RPC à des connexions SSH. Les Makefile pourraient être améliorés. On pourrait par ailleurs envisager des systèmes de RAID, de backup, et de versioning, pour assurer la sécurité des données.

Globalement, les services fonctionnent, mais peuvent être améliorés.

3 Bilan sur le projet

3.1 Etat de finalisation du projet

Salle

La salle d'expérimentation a beaucoup évolué au fil de l'année. A l'heure actuelle, elle se compose de 8 postes : 6 postes de tests reliés entre eux avec un réseau IPv6, un serveur netboot et un serveur de secours. Le serveur netboot propose des scripts permettant aux postes de tests d'être mis à jour facilement (Installation du noyau NetBSD, changement de librairie et de programmes de tests). Il permet aussi de changer facilement la configuration du switch et donc de la topologie. On a au final une salle opérationnelle comportant des outils de tests automatisés.

Protocole

La librairie dans sa version 3 est opérationnelle jusqu'à la construction de l'arbre. Le problème provient surtout de l'envoi du message entre la librairie et le driver. On suppose que celui-ci est dû à une mauvaise initialisation d'un des champs de longueur dans la structure d'envoi.

Le driver quant à lui n'a pas pu être testé convenablement. On ne peut envoyer un message qu'à partir de la librairie, or celle-ci comporte encore des problèmes. Les changements effectués concernant l'algorithme de routage dans le driver ont été testés unitairement avec succès en dehors du protocole. En ce qui concerne les modifications de structures de données et d'envoi vers les routeurs suivants, nous ne pouvons tester sans une librairie fonctionnelle à 100%.

3.2 Ce que le projet nous a apporté

3.2.1 Sur le plan du savoir technique

Le projet TBXcast étant un projet réseau, nous avons du en apprendre les bases avant même le module « Réseaux » du second semestre. Notre apprentissage s'est concentré sur la problématique du routage, domaine dans lequel nous avons peu à peu progressé. Le routage requiert des notions sur les graphes, et nous avons donc pu approfondir nos connaissances dans l'algorithmique des graphes.

La phase d'étude de Xcast et la phase de construction ont été pour nous une réelle confrontation au code de très bas niveau : noyau de NetBSD, programmation réseau. Peu familiers avec ce type de programmation, nous en avons peu à peu compris les principes et avons pu à la fois gérer la plateforme et réaliser à notre tour les modifications nécessaires dans le code du noyau de NetBSD.

Finalement, nous avons appris à manipuler la plateforme, c'est-à-dire nous familiariser avec le système d'exploitation NetBSD. Durant son installation et son utilisation, nous avons abordés les thèmes de la compilation, de la mise en réseau des postes, et de différents utilitaires comme par exemple les sniffer de paquets.

Ce projet fut donc une expérience très enrichissante sur le plan technique, car il nous a apporté à chacun des choses nouvelles et que nous n'avions abordées que très peu pendant le cursus de l'INSA. Bien que différant de projets plus « classiques » orientés objet, les connaissances acquises au cours de cette année donnent à notre profil une teinte intéressante.

3.2.2 Sur le plan organisationnel et relationnel

Le projet TBXcast a été une vraie simulation de projet dans un cadre professionnel. Les réunions hebdomadaires, les rapports et soutenances, et la méthodologie globale en ont fait une expérience très enrichissante. Chacun a pu, à tour de rôle, exercer une fonction de responsable. La mise en situation dans un projet de longue haleine fut donc une partie primordiale de l'enseignement de cette quatrième année. La bonne entente de l'équipe, sa synchronisation et son dynamisme ont fait de cette année de

projet un moment enrichissant et agréable. La motivation a succédé à des moments parfois difficiles, en général provoqués par les difficultés techniques rencontrées avec le protocole Xcast.

3.2.3 Perspectives

A ce jour, le projet TBXcast n'est pas terminé. L'envoi de paquet par la librairie doit être corrigé afin de permettre un fonctionnement correct des trois premières versions. La suite du développement de TBXcast, après la version 3, concernera l'implémentation de nouvelles fonctionnalités au protocole telles que la gestion de la qualité de service (QoS). Si le projet est reconduit l'année prochaine, nous léguons à nos successeurs une documentation riche et une plateforme de test opérationnelle. Nous leur conseillerons alors de dédier d'avantage de temps à la phase de réalisation.

4 Manuel et documentation

4.1 Description des fonctions principales de la librairie libtbxcast

Ceci est un résumé des fonctions nécessaires pour l'envoi d'un paquet par intermédiaire de TBXcast. Pour une documentation détaillée veuillez consulter la documentation Doxygen fournie.

TBXcastInitTopology

Initialise la topologie. Cette fonction doit être appelée au début de chaque application qui souhaite utiliser TBXcast.

TBXcastAddRoute

Ajoute une liaison entre deux nœuds dans la topologie réseau. Cette liaison est considérée à sens unique et peut contenir les informations sur la Qualité de Service.

TBXcastCreateRoutingMatrix

Calcule la matrice de routage. Cette matrice est nécessaire pour les calculs des arbres de routage et devrait être recalculé à chaque changement de topologie.

TBXcastCreateGroup

Crée un groupe TBXcast vide.

TBXcastAddMember

Ajoute un membre à un groupe TBXcast.

TBXcastCreateTree

Crée un arbre de routage à partir de la source vers les membres d'un groupe TBXcast.

TBXcastSetSockOpt

Initialise le socket de communication pour un groupe.

TBXcastAddTree

Associe un arbre de routage à un groupe. Cette arbre doit être d'abord calculé par TBXcastCreateTree ou fournit explicitement.

TBXcastSend

Envoie des données à un groupe TBXcast.

4.2 La plateforme d'expérimentation

La suite logicielle TBXplatform vise à une gestion optimale de la plateforme. Ses mécanismes d'automatisation sont simples à appréhender, sans être privés de la puissance indispensable pour les opérations de gestion.

4.2.1 Mécanisme de compilation : Makefile

Les Makefile servent à la compilation des sources de TBXcast. Le code est complètement organisé dans les sous-dossiers :

driver/*version*/ : Fichiers du driver (noyau) de la version *version.*

kernel/netbsd.*version* : Noyau compilé de la version *version.*

libtbxcast/*version*/ : Fichiers de la librairie de la version *version.*

tbxtest/*version*/ : Fichiers du programme de test de la version *version.*

tools/ : Fichiers des programmes de gestion *tbxpwr* et *tbxnet*.

bin/ : Dossier d'installation de la version courante de *tbxtest* et des programmes de gestion.

Les commandes du Makefile sont les suivantes :

make gen{build|install} VERSION=*version* : Compile/Installe le noyau, la librairie, et le programme de test de la version *version.*

make kern{build|install) VERSION=*version* : Compile/Installe le noyau de la version *version.*

make lib{build|install) VERSION=*version* : Compile/Installe la librairie de la version *version.*

make prog{build|install) VERSION=*version* : Compile/Installe le programme de test de la version *version.*

make tool{build|install) : Compile/Installe les programmes de gestion de la plateforme.

make help : Affiche l'aide du Makefile.

4.2.2 Gestion des machines : TBXpower

Le programme *tbxpwr* (TBXpower) contrôle l'état des machines de la plateforme. Il possède des commandes pour démarrer, arrêter, ou encore voir l'état des postes.

TBXpower permet notamment de simplifier le redémarrage de masse des machines après l'installation d'un nouveau noyau.

Commandes :

tbxpwr boot *hotes* : Démarre un ou plusieurs postes.

tbxpwr halt *hotes* : Arrête un ou plusieurs postes.

tbxpwr reboot *hotes* : Redémarre un ou plusieurs postes.

tbxpwr view : Liste l'etat des postes.

tbxpwr ident *hote* : Identifie un poste physiquement et de manière claire.

tbxpwr help : Affiche l'aide de la commande.

4.2.3 Gestion du réseau : TBXnet

Le programme *tbxnet* (TBXnet) gère l'état du réseau d'expérimentation. Il possède des commandes pour lier ou délier les postes, ou encore charger des topologies complètes.

TBXnet simplifie les tests en préparant la plateforme complète aux sessions de test grâce à des commandes simples.

Commandes :

tbxnet link *hotes* : Lie deux postes.

tbxnet unlink *hotes* : Delie deux postes.

tbxnet view : Liste l'état des postes.

tbxnet route : Etablit les routes entre tous les postes, à exécuter une fois que tous les liens sont établis.

tbxnet config *conf* : Charge une configuration reseau, et établit automatiquement les routes.

tbxnet unconfig : Supprime les liens entre tous les postes.

tbxnet help : Affiche l'aide de la commande.

Note pour les programmes *tbxpwr* et *tbxnet* :

hote : Chiffre compris entre 1 et le nombre de postes. Le chiffre représente le numéro de la machine auquel il est associé.

hotes : Se représente comme une liste d'intervalles d'un ou plusieurs hôtes, séparés par des virgules. Exemple : *1,3-5,7* représentera les hôtes 1,3,4,5 et 7.

conf : Chaine de caractères. Nom d'une configuration prédéfinie.

4.2.4 Manuel d'installation et d'utilisation de TBXcast

4.2.4.1 Installation du driver

Télécharger et extraire le code source du noyau à la racine :

```
tar -xzvf syssrc.tgz -C /
```

Patcher le noyau avec le code de TBXcast :

```
cp -r driver_tbxcast/* /usr/src/sys/
```

Installer le programme de configuration du noyau :

```
cd /usr/src/usr.bin/config && make && make install
```

Configurer le noyau :

```
cd /usr/src/sys/arch/i386/conf && config GENERIC_TBXCAST6
```

Compiler :

```
cd /usr/src/sys/arch/i386/compile/GENERIC_TBXCAST6/ && make depend && make
```

Installer le nouveau noyau :

```
cp /usr/src/sys/arch/i386/compile/GENERIC_TBXCAST6/netbsd /netbsd
```

Copier les headers :

```
cp /usr/src/sys/netinet6/{in6|tbxcast6}.h /usr/include/netinet6/
```

4.2.4.2 Installation de la librairie

Compiler et installer la librairie TBXcast :

```
cd lib_tbxcast/ && make && make install
```

4.2.4.3 Installation du programme de test

Compiler et installer le programme de test TBXcast :

```
cd prog_tbxcast/ && make && make install
```

4.2.4.4 Utilisation du programme de test

Redémarrer une fois le noyau installé :

```
reboot
```

Configurer les routes pour l'interface TBXcast :

```
ifconfig tbxcst0 up

route add -host -inet6 ff15::20 ::1

route change -host -inet6 ff15::20 -ifp tbxcst0
```

Activer le forwarding ipv6 :

```
sysctl -w net.inet6.ip6.forwarding=1
```

Configurer le réseau IPv6, sur chaque poste (adapter en fonction de la topologie souhaitée) :

```
ifconfig rtk0 inet6 a::1
```

Tester le protocole (Séparer plusieurs destinataires par des virgules) :

```
tbxtest a::1
```

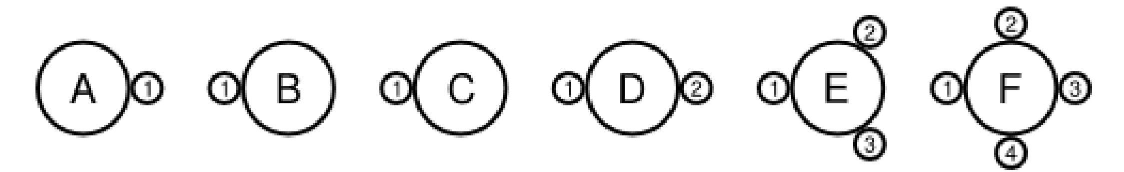
5 Annexe

5.1 Topologies de test

Afin de tester notre protocole nous avons crées plusieurs topologies du réseau dont chacune permet de tester une situation particulière dans le réseau.

Configuration des machines

Nous sommes en possession de 6 machines et 12 cartes réseau pour construire notre platforme de test. Afin de pouvoir exploiter un maximum de configurations possibles nous avons fait le choix de configurer les machines de la façon suivante :

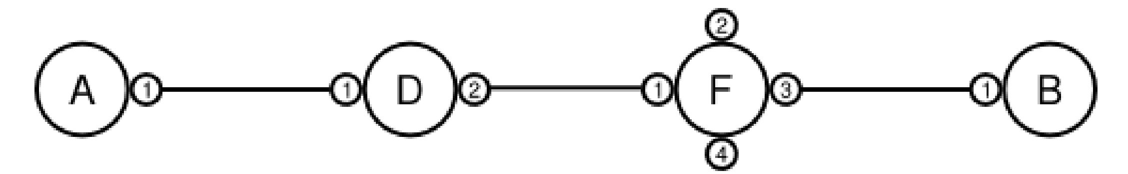


- 3 machines avec 1 carte réseau chacune
- 1 machine avec 2 cartes réseau
- 1 machine avec 3 cartes réseau
- 1 machine avec 4 cartes réseau

Les machines avec 1 carte réseau servent des sources et destinations, les machines possédant plusieurs cartes réseau servent de routeurs.

Topologie 1

Réseau simple sans branchement

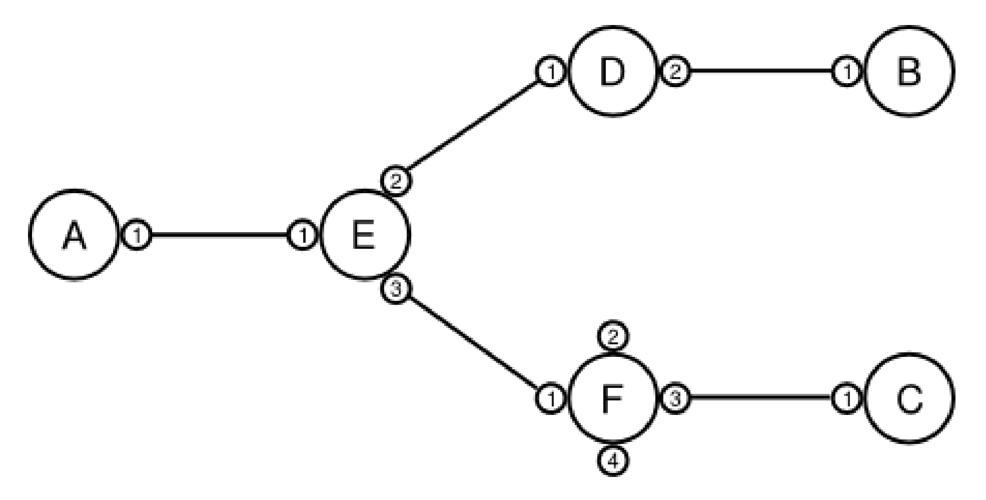


Cette topologie est très simple et teste l'envoi d'un paquet en direct. Elle permet de vérifier le fonctionnement du protocole dans une situation basique.

Tests associés

- Envoi d'un paquet depuis A vers D
  - Teste que le protocole fonctionne
- Envoi d'un paquet depuis A vers B
  - Teste le passage de paquet aux routeurs suivant dans le driver TBXcast
- Envoi d'un paquet depuis A vers D, F et B
  - Vérifie la gestion des routeurs marqués en tant que destinations

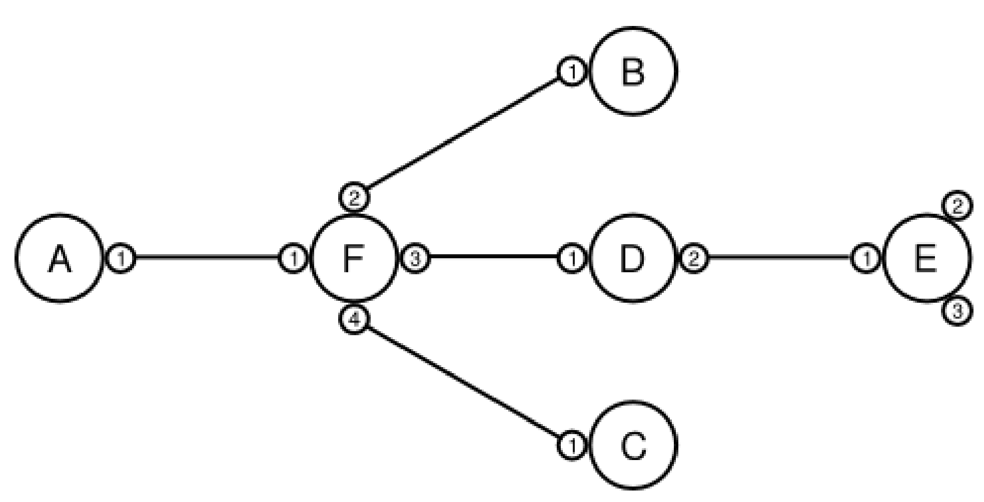
Topologie 2
Réseau simple avec un routeur en étoile



Tests associés

- Envoi d'un paquet depuis A vers B
  - Vérifie le bon choix d'interface de sortie dans un routeur
- Envoi d'un paquet depuis A vers B et C
  - Teste la duplication du paquet dans le routeur
- Envoi d'un paquet depuis A vers B, F et C
  - Teste à la fois le branchement et la gestion des routeurs en tant que destinations

Topologie 3
Réseau contenant un routeur avec quatre interfaces associées

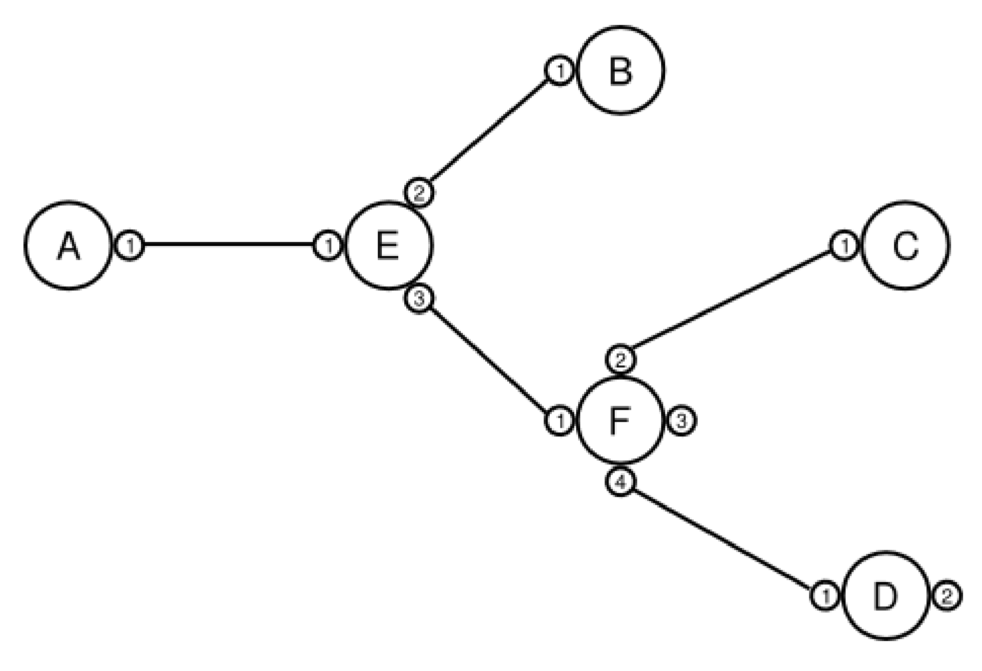


Tests associés

- Envoi d'un paquet depuis A vers B et C
  - Vérifie la sélection des bonnes interfaces de sortie
- Envoi d'un paquet depuis A vers B, E et C
  - Teste le bon découpage de l'arbre dans le cas de triple branchement avec des tailles de sous-arbres inégales
- Envoi d'un paquet depuis E vers A, B et C
  - Vérifie le branchement après un renvoi direct

Topologie 4

Réseau possédant deux nœuds de branchement



Tests associés

- Envoi d’un paquet depuis A vers C
  - Teste le choix de l’interface de sortie dans le driver
- Envoi d’un paquet depuis A vers B, C et D
  - Teste le traitement de l’arbre de routage lors d’un double branchement
- Envoi d’un paquet depuis A vers B, E, F et C
  - Test complet du protocole. Toutes les fonctions de TBXcast sont sollicitées.

6 Références

Site du projet : http://tbxcast.xipp.net

Page Wiki du projet : http://tbxcast.xipp.net/wiki

Documentation Doxygen de la librairie : http://tbxcast.xipp.net/doc/libtbxcast/